Cryptographic Hash Functions

Objectives

- **To introduce general ideas behind cryptographic hash functions**
- **To discuss the Merkle-Damgard scheme as the basis for iterated hash functions**
- **To discuss structure of MD5 algorithm**
- **To discuss structure of SHA algorithm**

INTRODUCTION

- A cryptographic hash function takes a message of arbitrary length and creates a message digest of fixed length.
- Was originally proposed to generate input to digital signatures.

Desirable features: preimage resistant, second preimage resistant and collision resistant.

Iterated Hash Function

- All cryptographic hash functions need to create a fixed size digest out of a variable size message.
- Creating such a function is best accomplished using concept of iteration.
- Instead of using a hash function with variable size input, a function with fixed size input is created and is used a necessary number of times.

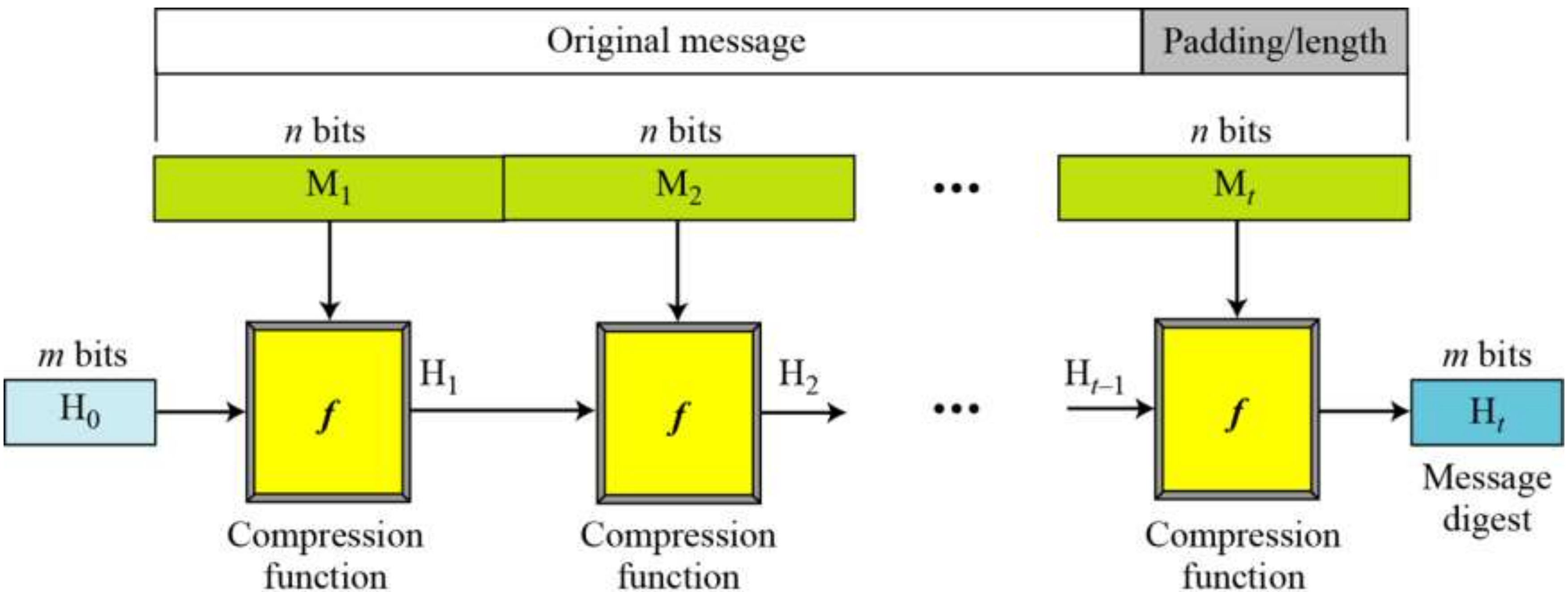
Compression Function

- The fixed size input function is referred to as a compression function.
- It compresses an n-bit string to create m-bit string where n is normally greater than m.
- This scheme is referred to as an Iterated Cryptographic Hash Function.

Iterated Hash Function

Merkle-Damgard Scheme : is collision resistant if compression function is collision resistant.

Merkle-Damgard scheme



Merkle-Damgard Scheme

The scheme uses the following steps:

1. The message length and padding are appended to the message to create an augmented message that can be evenly divided into blocks of n bits, where n is the size of the block to be processed by the compression function.
2. The message is then considered as t blocks, each of n bits. We call each block M_1, $M_2, \ldots, M_t$. We call the digest created at t iterations $H_1, H_2, \ldots, H_t$.
3. Before starting the iteration, the digest H_0 is set to a fixed value, normally called IV (initial value or initial vector).
4. The compression function at each iteration operates on H_{i-1} and M_i to create a new H_i. In other words, we have $H_i = f(H_{i-1}, M_i)$, where f is the compression function.
5. H_t is the cryptographic hash function of the original message, that is, $h(M)$.

Two Groups of Compression Functions

1. The compression function is made from scratch.
2. A symmetric-key block cipher serves as a compression function.

Hash Functions made from scratch

1. Message Digest (MD)

a) Message digest (MD) referred as MD2, MD4 and MD5 were designed by Ron Rivest.

b) MD5 is strengthened version.

2. Secure Hash Algorithm (SHA)

a) Is a standard that was developed by NIST and published in FIPS 180.

b) It is sometimes referred to as SHS, mostly based on MD5

MD5

- MD5 algorithm was developed by Professor Ronald L. Rivest in 1991.
- According to RFC 1321, “MD5 message-digest algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input.
- The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.”

Steps in MD5

1. Append padding bits
2. Append length
3. Initialize MD buffer
4. Process message in 16-word blocks
5. Output

MD5

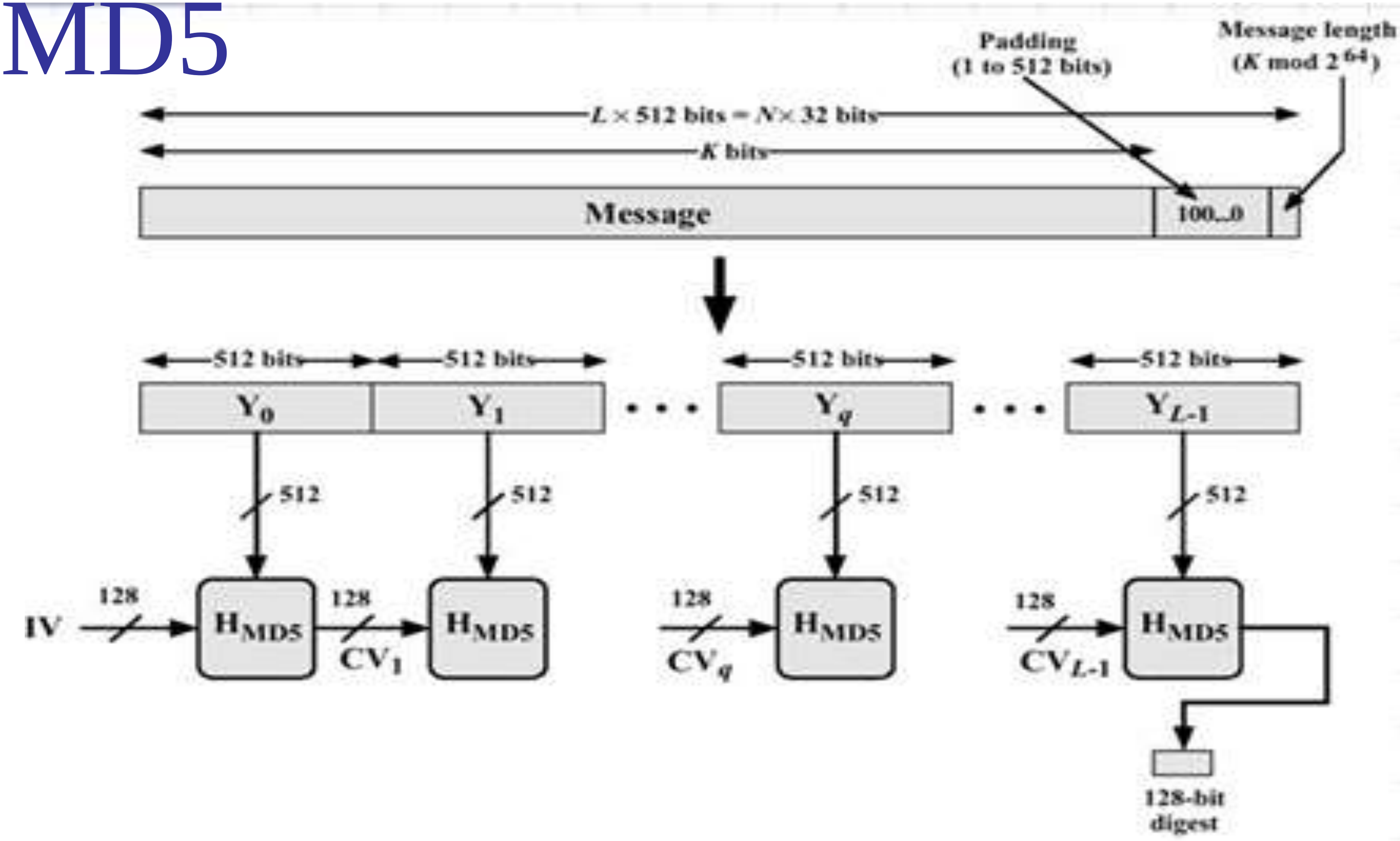
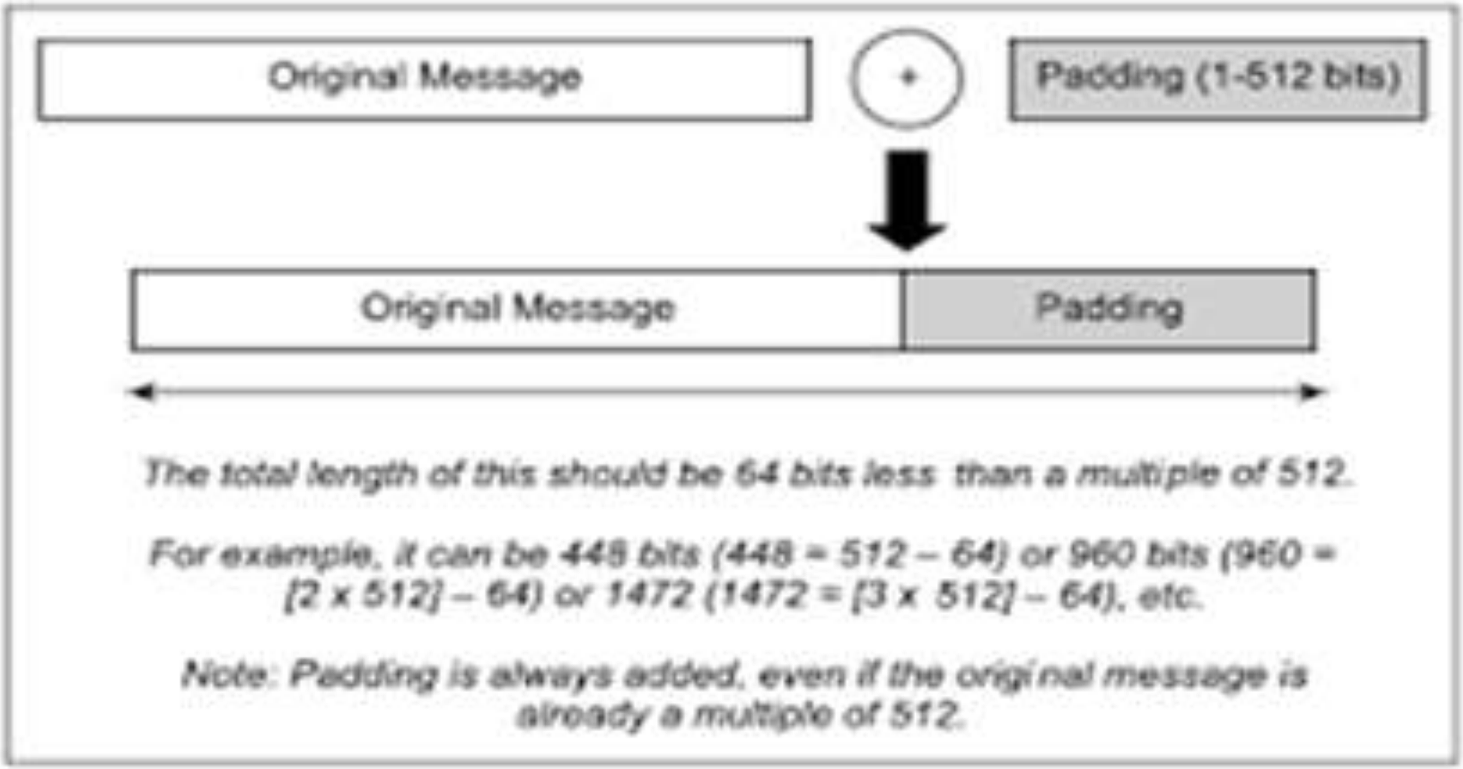


Figure 9.1 Message Digest Generation Using MD5

Step 1: append padding bits

- The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long.
- Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512.


Original Message
+
Padding (1-512 bits)
Original Message
Padding
The total length of this should be 64 bits less than a multiple of 512.
For example, it can be 448 bits (448 = 512 – 64) or 960 bits (960 = [2 x 512] – 64) or 1472 (1472 = [3 x 512] – 64), etc.
Note: Padding is always added, even if the original message is already a multiple of 512.

Step 2: append length

- A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step.

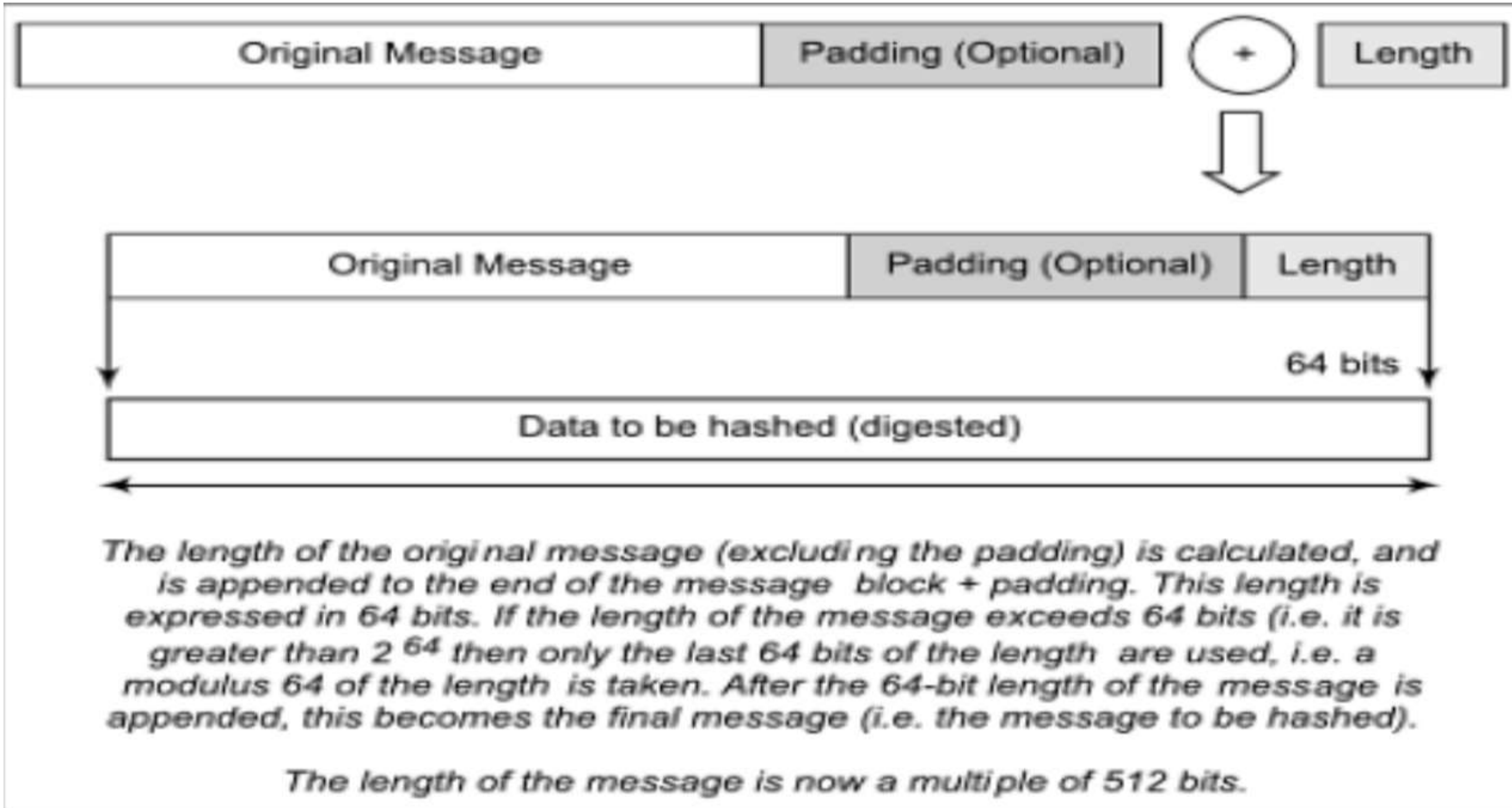


Figure 2 : Append Length

After Step 1 and Step 2

Preparing the input

- each block (512 bits) is divided into 16 words of 32 bits each.
- These are denoted as $\mathbf{M_0 \ldots M_{15}}$

Step 3: Initialize MD buffer

- MD5 uses a buffer that is made up of four **words** that are each 32 bits long. These words are called A, B, C and D. They are initialized as:

    word A: 01 23 45 67

    word B: 89 ab cd ef

    word C: fe dc ba 98

    word D: 76 54 32 10

Step 4: process message in 16 words block

- We first define four auxiliary functions for four rounds that each take as input three 32-bit words and produce as output one 32-bit word.

  F(X,Y,Z) = (X and Y) or (not(X) and Z)
  G(X,Y,Z) = (X and Z) or (Y and not(Z))
  H(X,Y,Z) = X xor Y xor Z
  I(X,Y,Z) = Y xor (X or not(Z))

They apply the logical operators and, or, not and xor to the input bits.

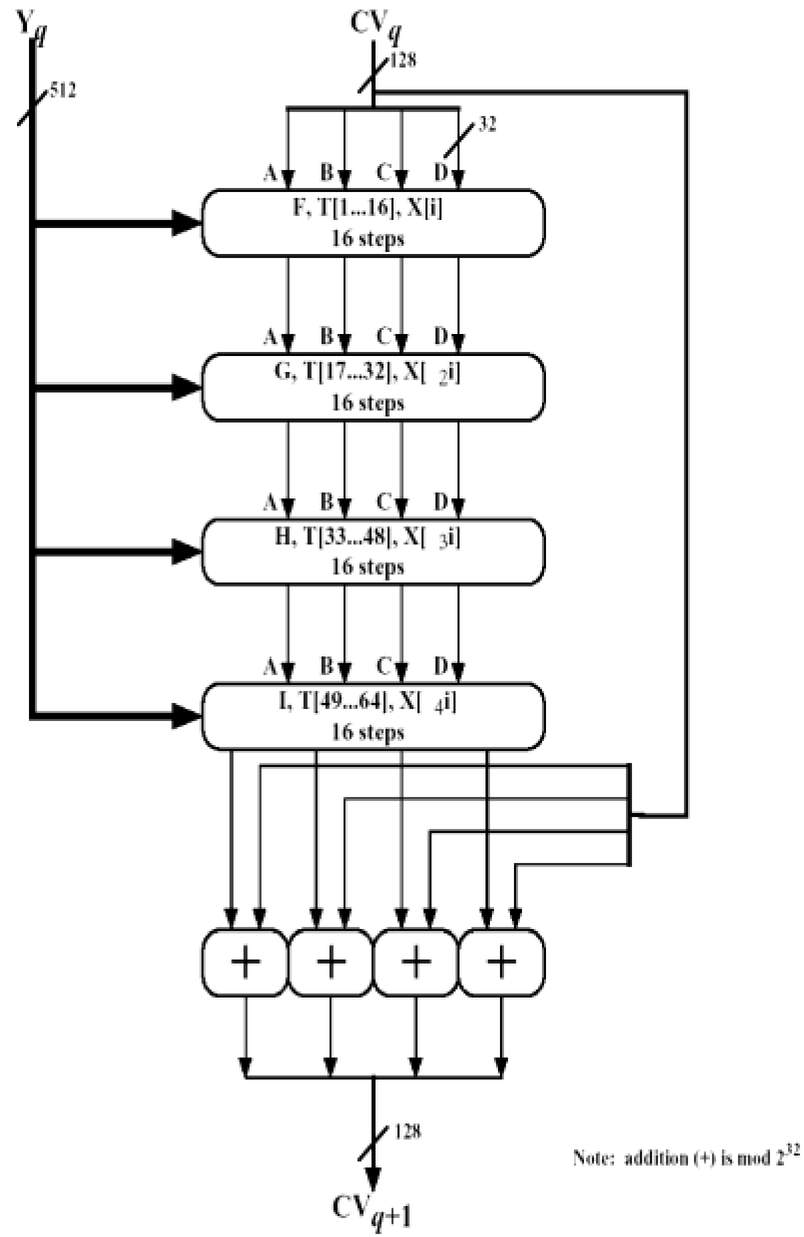


Figure 9.2 MD5 Processing of a Single 512-bit Block (MD5 Compression Function)

MD5 Compression Function

- each round has 16 steps of the form:

  $$a = b+((a+g(b,c,d)+X[k]+T[i])<<<s)$$

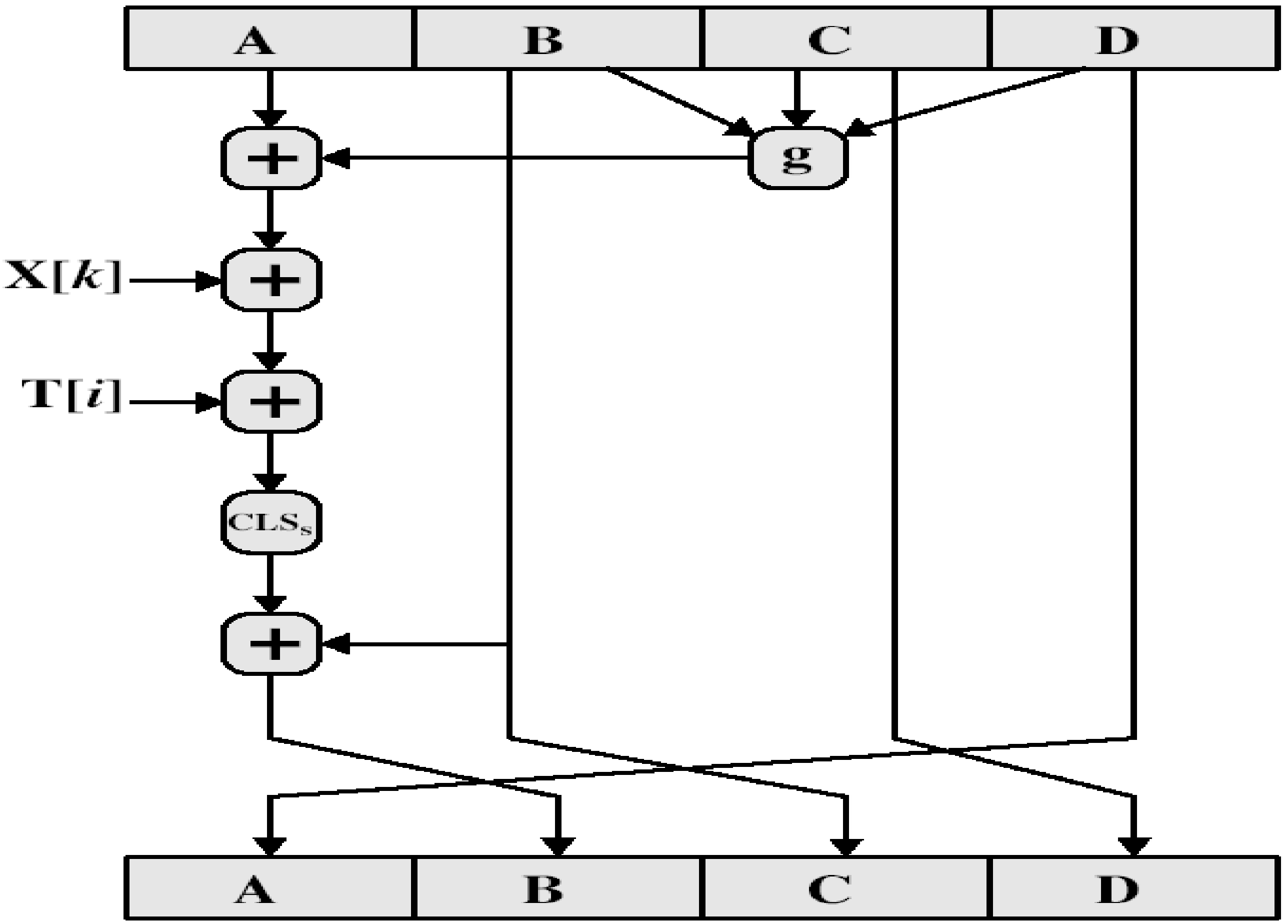
- a,b,c,d refer to the 4 words of the buffer, but used in varying permutations
  - note this updates 1 word only of the buffer
  - after 16 steps each word is updated 4 times

Continued…

- where g(b,c,d) is a different nonlinear function in each round (F,G,H,I)
- X[k] is the kth 32-bit word in the current message block
- T[i] denote the i-th element of the table. T[1 ... 64] constructed from the sin function, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians.
- The item "<<<s" denotes a binary left shift by *s* bits

Continued...

$$a = b+((a+g(b,c,d)+X[k]+T[i])<<<s)$$



Iteration	a	b	c	d	M	s	t
1	a	b	c	d	M[0]	7	t[1]
2	d	a	b	c	M[1]	12	t[2]
3	c	d	a	b	M[2]	17	t[3]
4	b	c	d	a	M[3]	22	t[4]
5	a	b	c	d	M[4]	7	t[5]
6	d	a	b	c	M[5]	12	t[6]
7	c	d	a	b	M[6]	17	t[7]
8	b	c	d	a	M[7]	22	t[8]
9	a	b	c	d	M[8]	7	t[9]
10	d	a	b	c	M[9]	12	t[10]
11	c	d	a	b	M[10]	17	t[11]
12	b	c	d	a	M[11]	22	t[12]
13	a	b	c	d	M[12]	7	t[13]
14	d	a	b	c	M[13]	12	t[14]
15	c	d	a	b	M[14]	17	t[15]
16	b	c	d	a	M[15]	22	t[16]

. 4.35 (a) Round 1

Iteration	a	B	c	d	M	s	t
1	a	b	c	d	M[1]	5	t[17]
2	d	a	b	c	M[6]	9	t[18]
3	c	d	a	b	M[11]	14	t[19]
4	b	c	d	a	M[0]	20	t[20]
5	a	b	c	d	M[5]	5	t[21]
6	d	a	b	c	M[10]	9	t[22]
7	c	d	a	b	M[15]	14	t[23]
8	b	c	d	a	M[4]	20	t[24]
9	a	b	c	d	M[9]	5	t[25]
10	d	a	b	c	M[14]	9	t[26]
11	c	d	a	b	M[3]	14	t[27]
12	b	c	d	a	M[8]	20	t[28]
13	a	b	c	d	M[13]	5	t[29]
14	d	a	b	c	M[2]	9	t[30]
15	c	d	a	b	M[7]	14	t[31]
16	b	c	d	a	M[12]	20	t[32]

Fig. 4.35 (b) Round 2

Iteration	a	b	c	d	M	s	t
1	a	b	c	d	M[5]	4	t[33]
2	d	a	b	c	M[8]	11	t[34]
3	c	d	a	b	M[11]	16	t[35]
4	b	c	d	a	M[14]	23	t[36]
5	a	b	c	d	M[1]	4	t[37]
6	d	a	b	c	M[4]	11	t[38]
7	c	d	a	b	M[7]	16	t[39]
8	b	c	d	a	M[10]	23	t[40]
9	a	b	c	d	M[13]	4	t[41]
10	d	a	b	c	M[0]	11	t[42]
11	c	d	a	b	M[3]	16	t[43]
12	b	c	d	a	M[6]	23	t[44]
13	a	b	c	d	M[9]	4	t[45]
14	d	a	b	c	M[12]	11	t[46]
15	c	d	a	b	M[15]	16	t[47]
16	b	c	d	a	M[2]	23	t[48]

Iteration	a	b	c	d	M	s	t
1	a	b	c	d	M[0]	6	t[49]
2	d	a	b	c	M[7]	10	t[50]
3	c	d	a	b	M[14]	15	t[51]
4	b	c	d	a	M[5]	21	t[52]
5	a	b	c	d	M[12]	6	t[53]
6	d	a	b	c	M[3]	10	t[54]
7	c	d	a	b	M[10]	15	t[55]
8	b	c	d	a	M[1]	21	t[56]
9	a	b	c	d	M[8]	6	t[57]
10	d	a	b	c	M[15]	10	t[58]
11	c	d	a	b	M[6]	15	t[59]
12	b	c	d	a	M[13]	21	t[60]
13	a	b	c	d	M[4]	6	t[61]
14	d	a	b	c	M[11]	10	t[62]
15	c	d	a	b	M[2]	15	t[63]
16	b	c	d	a	M[9]	21	t[64]

t[i]	Value	t[i]	Value	t[i]	Value	t[i]	Value
t[1]	D76AA478	t[17]	F61E2562	t[33]	FFFA3942	t[49]	F4292244
t[2]	E8C7B756	t[18]	C040B340	t[34]	8771F681	t[50]	432AFF97
t[3]	242070DB	t[19]	265E5A51	t[35]	699D6122	t[51]	AB9423A7
t[4]	C1BDCEEE	t[20]	E9B6C7AA	t[36]	FDE5380C	t[52]	FC93A039
t[5]	F57C0FAF	t[21]	D62F105D	t[37]	A4BEEA44	t[53]	655B59C3
t[6]	4787C62A	t[22]	02441453	t[38]	4BDECFA9	t[54]	8F0CCC92
t[7]	A8304613	t[23]	D8A1E681	t[39]	F6BB4B60	t[55]	FFEFF47D
t[8]	FD469501	t[24]	E7D3FBC8	t[40]	BEBFBC70	t[56]	85845DD1
t[9]	698098D8	t[25]	21E1CDE6	t[41]	289B7EC6	t[57]	6FA87E4F
t[10]	8B44F7AF	t[26]	C33707D6	t[42]	EAA127FA	t[58]	FE2CE6E0
t[11]	FFFF5BB1	t[27]	F4D50D87	t[43]	D4EF3085	t[59]	A3014314
t[12]	895CD7BE	t[28]	455A14ED	t[44]	04881D05	t[60]	4E0811A1
t[13]	6B901122	t[29]	A9E3E905	t[45]	D9D4D039	t[61]	F7537E82
t[14]	FD987193	t[30]	FCEFA3F8	t[46]	E6DB99E5	t[62]	BD3AF235
t[15]	A679438E	t[31]	676F02D9	t[47]	1FA27CF8	t[63]	2AD7D2BB
t[16]	49B40821	t[32]	8D2A4C8A	t[48]	C4AC5665	t[64]	EB86D391

Fig. 4.36 Values of the table *t*

T[i]

```
for i from 0 to 63
    K[i] := floor(2^32 × abs(sin(i + 1)))
end for
//(Or just use the following precomputed table):
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }
```

Step 5: Output

- After all rounds have been performed, the buffers A, B, C and D contain the MD5 digest of the original input.

MD4

- precursor to MD5
- also produces a 128-bit hash of message
- has 3 rounds of 16 steps versus 4 in MD5
- design goals:
  - collision resistant (hard to find collisions)
  - fast, simple, compact

MD5 vs. MD4

1. A fourth round has been added.
2. Each step has a unique additive constant.
3. The function g in round 2 was changed from (XY v XZ v YZ) to (XZ v Y not(Z)).
4. Each step adds in the result of the previous step.
5. The order in which input words are accessed in rounds 2 and 3 is changed.
6. The shift amounts in each round have been optimized. The shifts in different rounds are distinct.

Summary

- Comparing to other digest algorithms, MD5 is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length.
- It performs very fast on 32-bit machine.
- MD5 is being used heavily from large corporations, such as IBM, Cisco Systems, to individual programmers.
- MD5 is considered one of the most efficient algorithms currently available.

Secure Hash Algorithm (SHA-1)

- SHA was designed by NIST in 1993, revised 1995 as SHA-1
- The algorithm is SHA, the standard is SHS
- produces 160-bit hash values
- now the generally preferred hash algorithm
- based on design of MD4 with key differences

Characteristics of SHA algorithms

Table 12.1 *Characteristics of Secure Hash Algorithms (SHAs)*

Characteristics	*SHA-1*	*SHA-224*	*SHA-256*	*SHA-384*	*SHA-512*
Maximum Message size	$2^{64} - 1$	$2^{64} - 1$	$2^{64} - 1$	$2^{128} - 1$	$2^{128} - 1$
Block size	512	512	512	1024	1024
Message digest size	160	224	256	384	512
Number of rounds	80	64	64	80	80
Word size	32	32	32	64	64

Steps in SHA

1. Append padding bits
2. Append length
3. Initialize MD buffer
4. Process message in 16-word blocks
5. Output

SHA-1 Overview

Step 1: Pad message so its length is 448 mod 512

Step 2: Append a 64-bit length value to message

Step 3: Initialise 5-word (160-bit) buffer (A,B,C,D,E)

Step 4: Process message in 16-word (512-bit) chunks:

- expand 16 words into 80 words by mixing & shifting
- add output to input buffer to form new buffer value

Step 5: Output hash value is the final buffer value

Processing of SHA-512

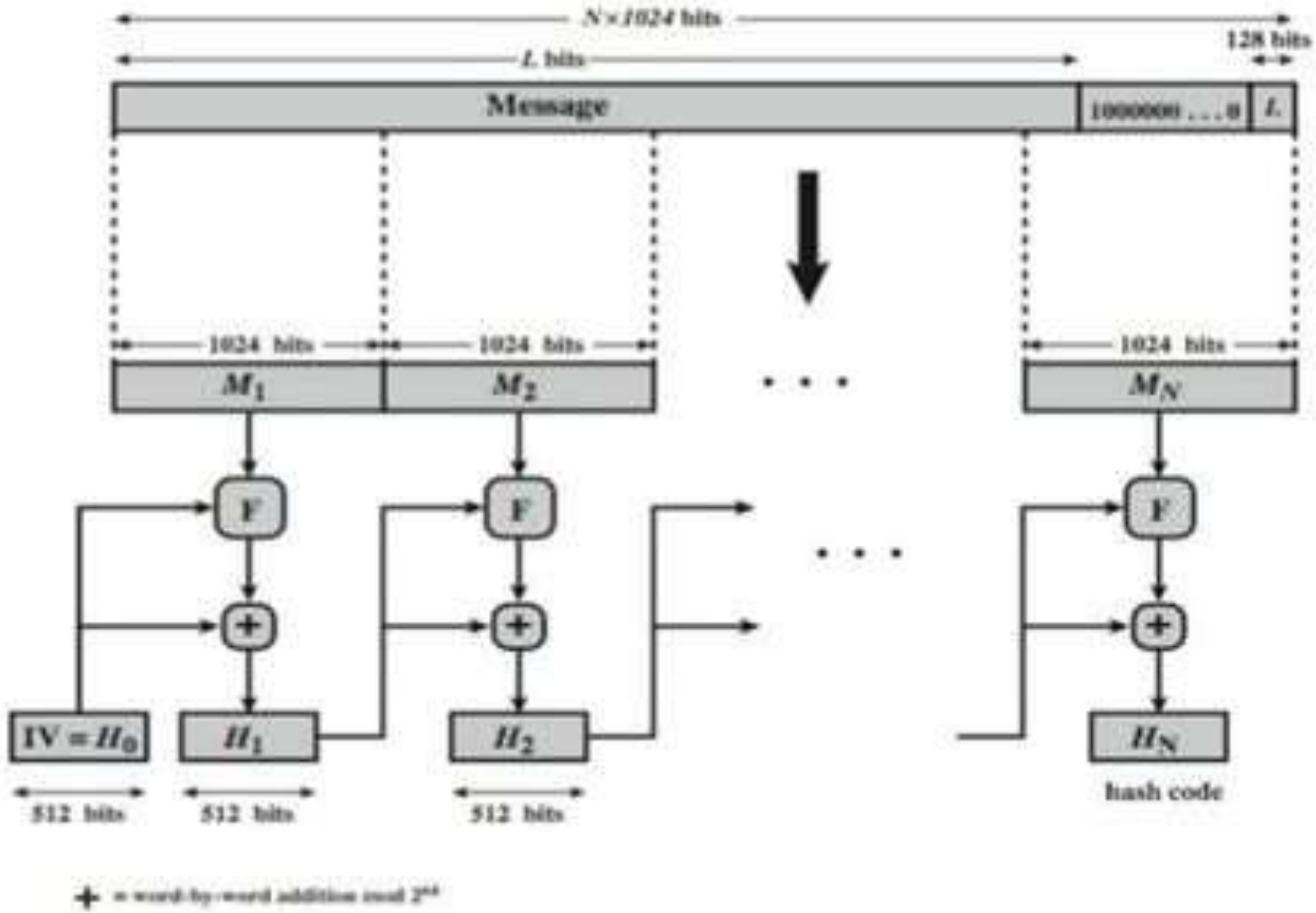


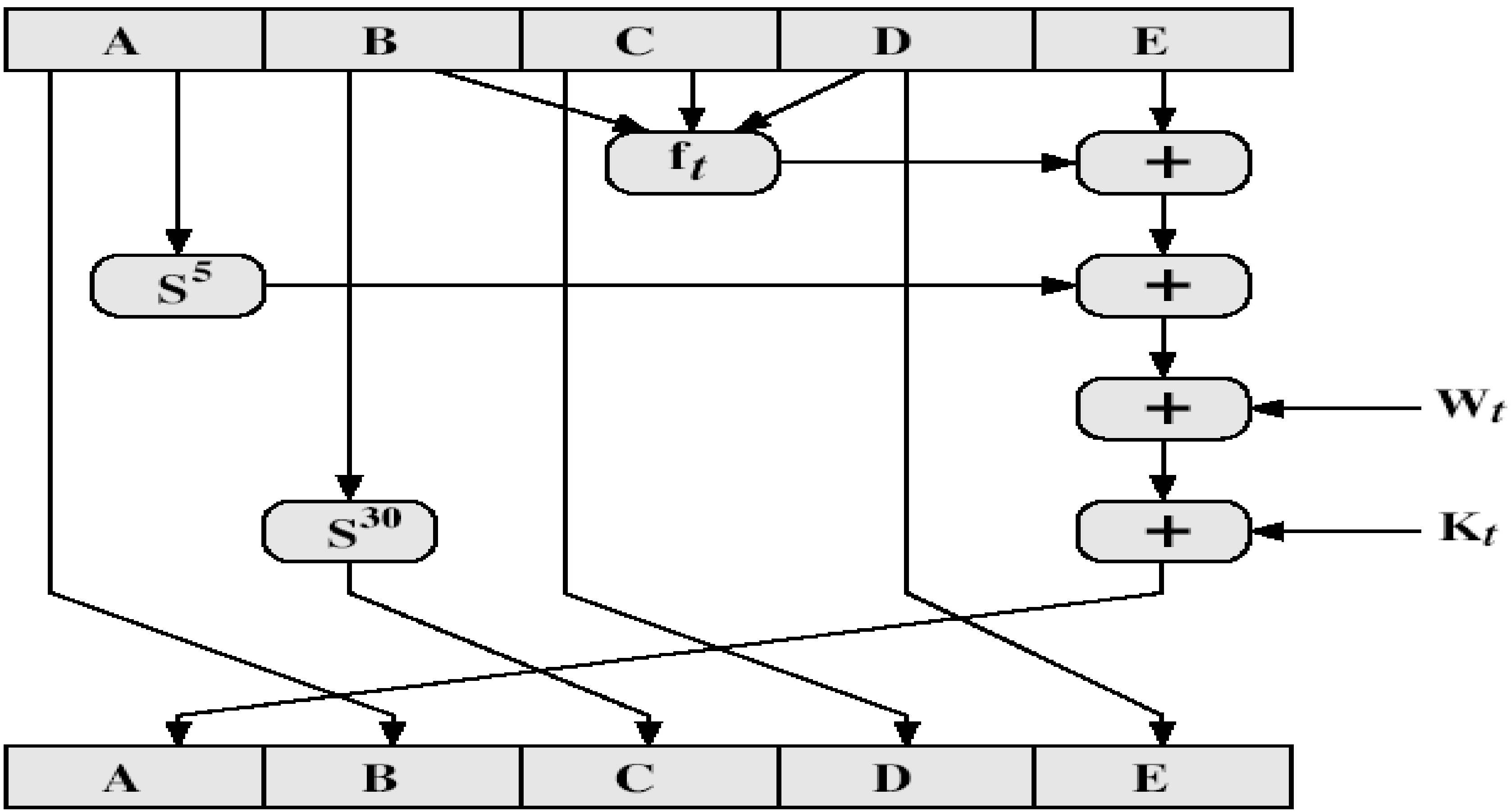
Figure: SHA-512 Processing of a Single 1024-Bit Block 4

SHA-1 Compression Function

- each round replaces the 5 buffer words thus:

  $(A,B,C,D,E) <\text{-}(E+f(t,B,C,D)+(A<<5)+W_t+K_t),A,(B<<30),C,D)$
- a,b,c,d,e refer to the 5 words of the buffer
- t is the step number
- f(t,B,C,D) is nonlinear function for round
- W_t is derived from the message block
- K_t is a constant value derived from sin function

SHA-1 Compression Function



SHA-1 verses MD5

- brute force attack is harder (160 vs 128 bits for MD5)
- not vulnerable to any known attacks (compared to MD4/5)
- a little slower than MD5
- both designed as simple and compact

Revised Secure Hash Standard

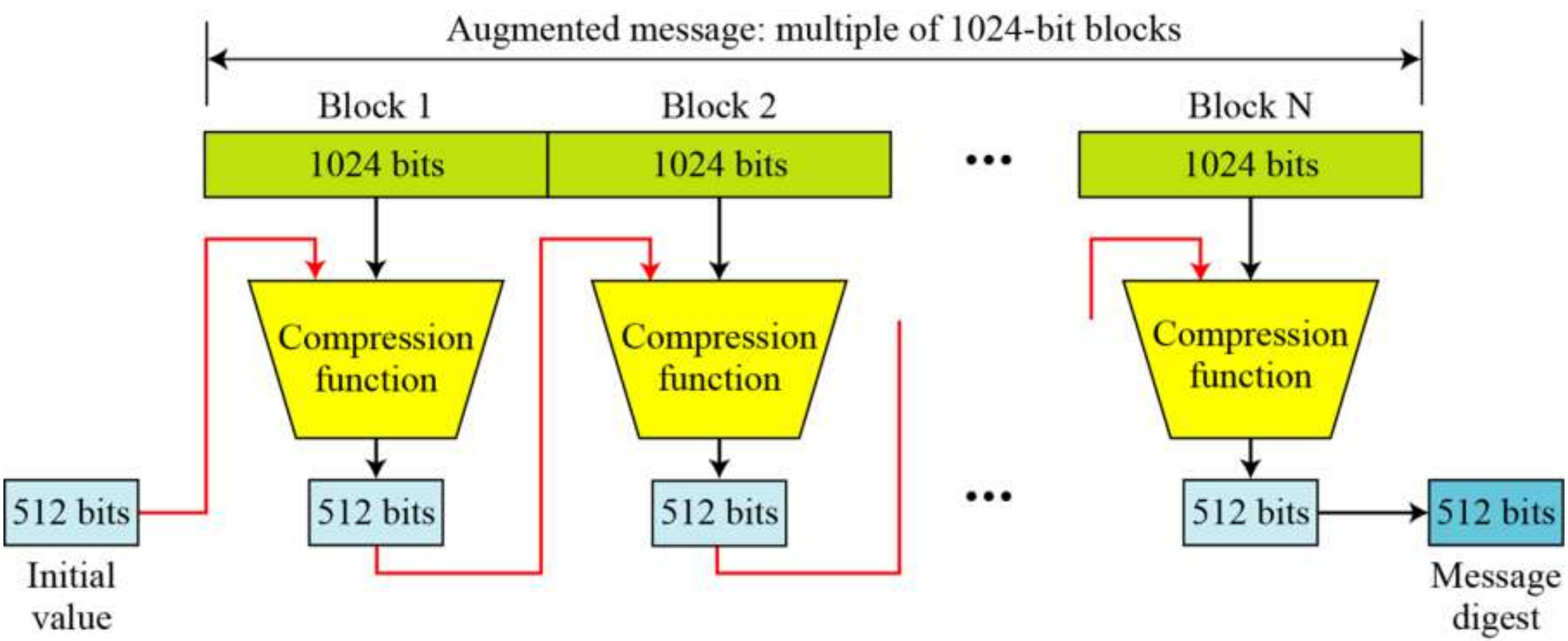
- NIST has issued a revision FIPS 180-2
- adds additional hash algorithms
- SHA-224, SHA-256, SHA-384, SHA-512
- designed for compatibility with increased security provided by the AES cipher
- structure & detail is similar to SHA-1
- hence analysis should be similar

SHA-512

- SHA-512 is the version of SHA with a 512-bit message digest.
- This version, like the others in the SHA family of algorithms, is based on the Merkle-Damgard scheme.

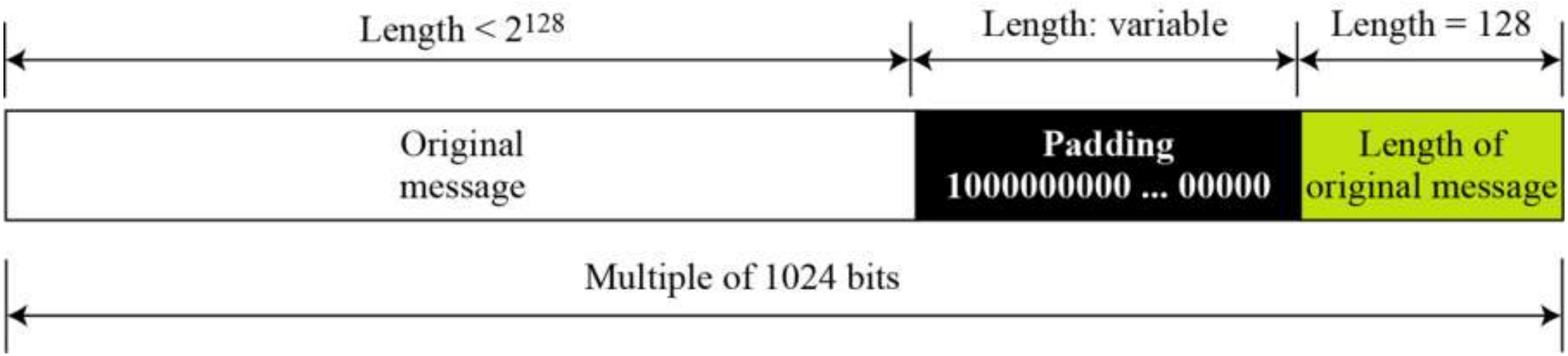
Introduction

Figure ***Message digest creation SHA-512***



Step 1&2: Padding and append length

Figure ***Padding and length field in SHA-512***



Message Preparation

SHA-512 insists that the length of the original message be less than 2^{128} bits.

Continued

Example

What is the number of padding bits if the length of the original message is 2590 bits?

Solution

We can calculate the number of padding bits as follows:

$$|P| = (-2590 - 128) \bmod 1024 = -2718 \bmod 1024 = 354$$

The padding consists of one 1 followed by 353 0's.

Step 3: Initialize MD buffer

- SHA-512is word oriented. Word is defined as 64 bits.
- Each block of message consists of 16 64-bit words.
- Message digest is also made of 64-bit words and hence message digest is 8 words (8*64=512 bits).
- 8 words named as A,B,C,D,E,G,H

Step 3: Initialize MD buffer

A message block and the digest as words

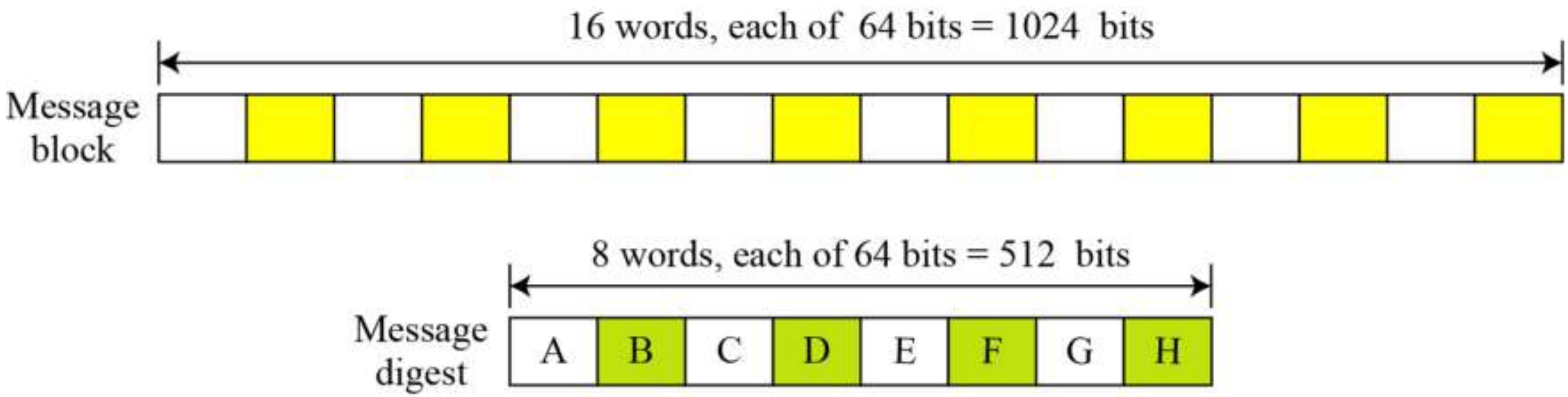


Table 12.2 *Values of constants in message digest initialization of SHA-512*

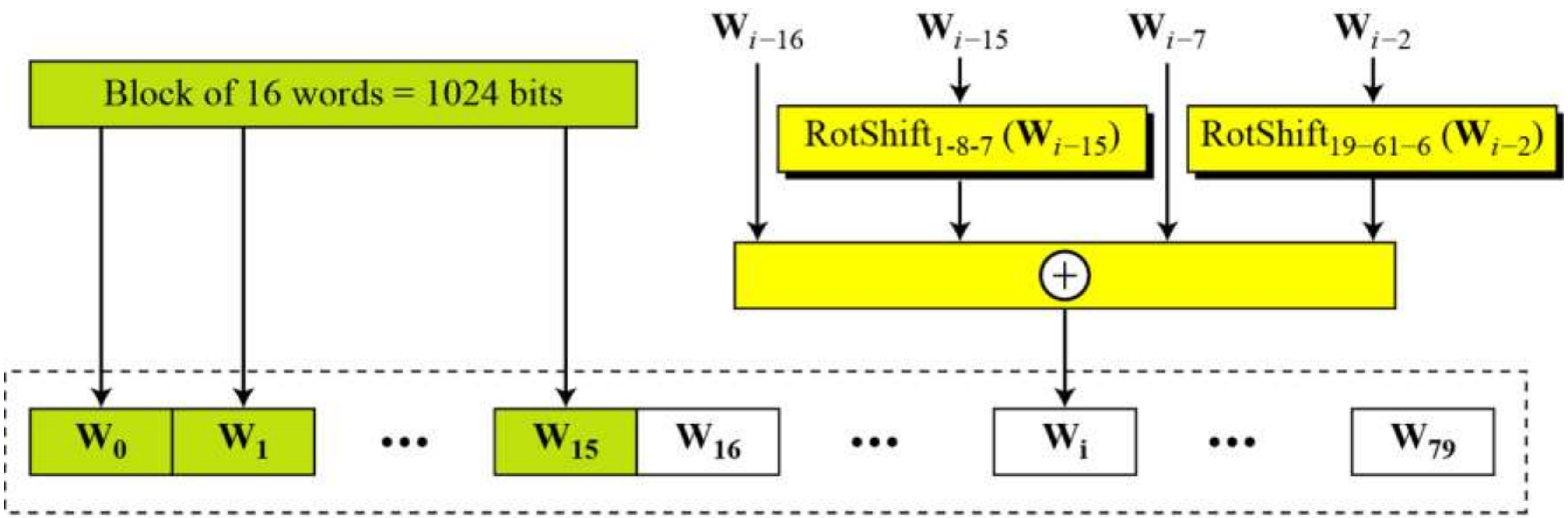
Buffer	*Value (in hexadecimal)*	*Buffer*	*Value (in hexadecimal)*
A_0	**6A09E667F3BCC908**	E_0	**510E527FADE682D1**
B_0	**BB67AE8584CAA73B**	F_0	**9B05688C2B3E6C1F**
C_0	**3C6EF372EF94F828**	G_0	**1F83D9ABFB41BD6B**
D_0	**A54FE53A5F1D36F1**	H_0	**5BE0CD19137E2179**

Word Expansion

- Before processing, each message block must be expanded.
- Each block is made of 16 words of 64 bits each.(1024 bits)
- In the next step i.e. processing step, we need 80 words.
- So 16 word block need to be expanded to 80 words, from W_0 to W_{79}.

Word Expansion

Figure ***Word expansion in SHA-512***



$\text{RotShift}_{l\text{-}m\text{-}n}(x)$: $\text{RotR}_l(x) \oplus \text{RotR}_m(x) \oplus \text{ShL}_n(x)$

$\text{RotR}_i(x)$: Right-rotation of the argument x by i bits
$\text{ShL}_i(x)$: Shift-left of the argument x by i bits and padding the left by 0's.

Example

Show how W60 is made.

Solution

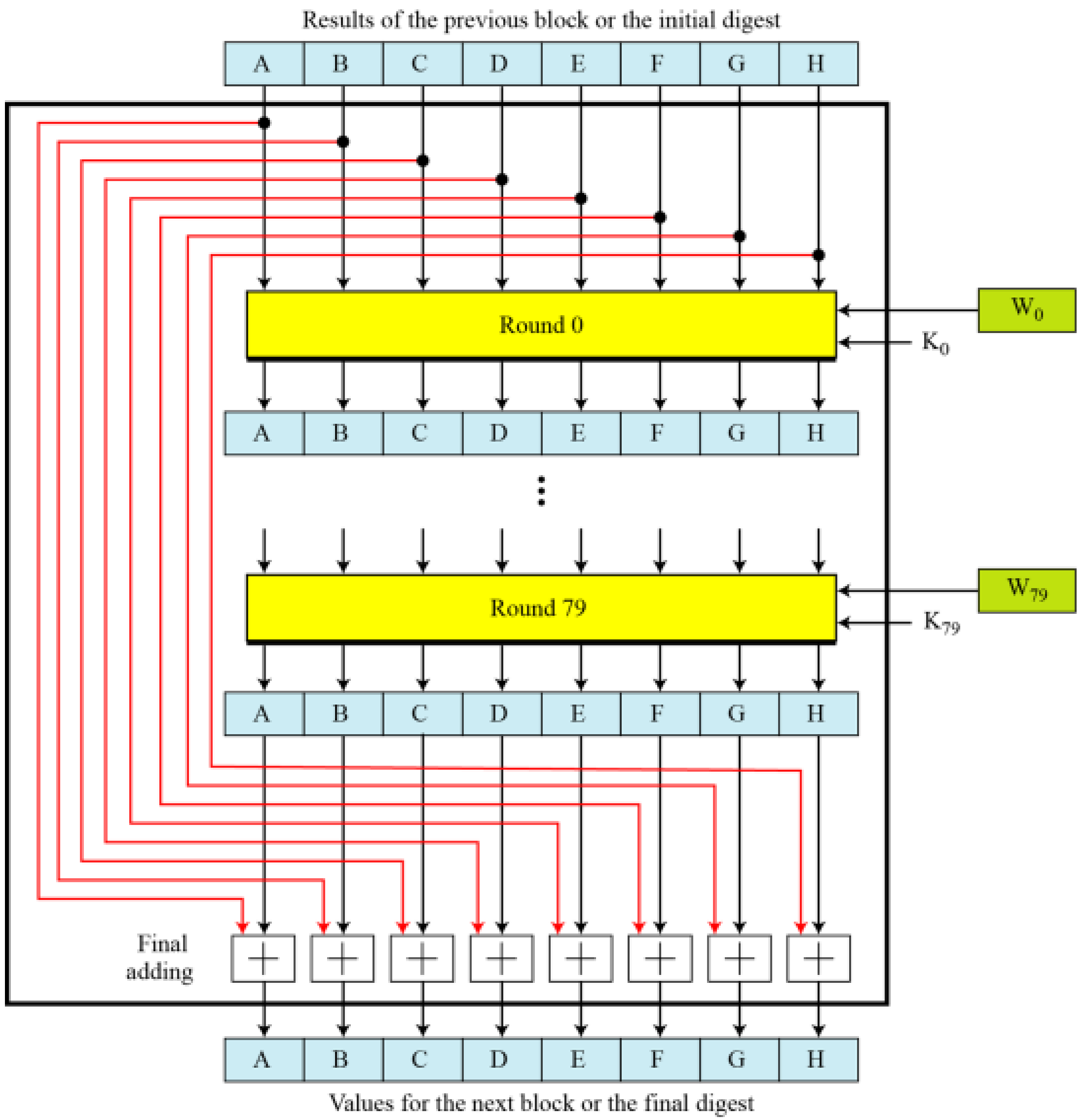
Each word in the range W16 to W79 is made from four previously-made words. W60 is made as

$$W_{60} = W_{44} \oplus \text{RotShift}_{1\text{-}8\text{-}7}\,(W_{45}) \oplus W_{53} \oplus \text{RotShift}_{19\text{-}61\text{-}6}\,(W_{58})$$

Step 4: Process message

- Processing of each block of data (1024 bits) in SHA-512 involves 80 rounds.
- In each round, contents of eight previous buffers, one word from expanded block (W_i), and one 64 bit constant (K_i) are mixed together and then operated on to create a new set of eight buffers.
- At the beginning, values of 8 buffers are saved into temporary variables and at the end of processing, these values are added to the values created from last round. This last operation is *Final adding*

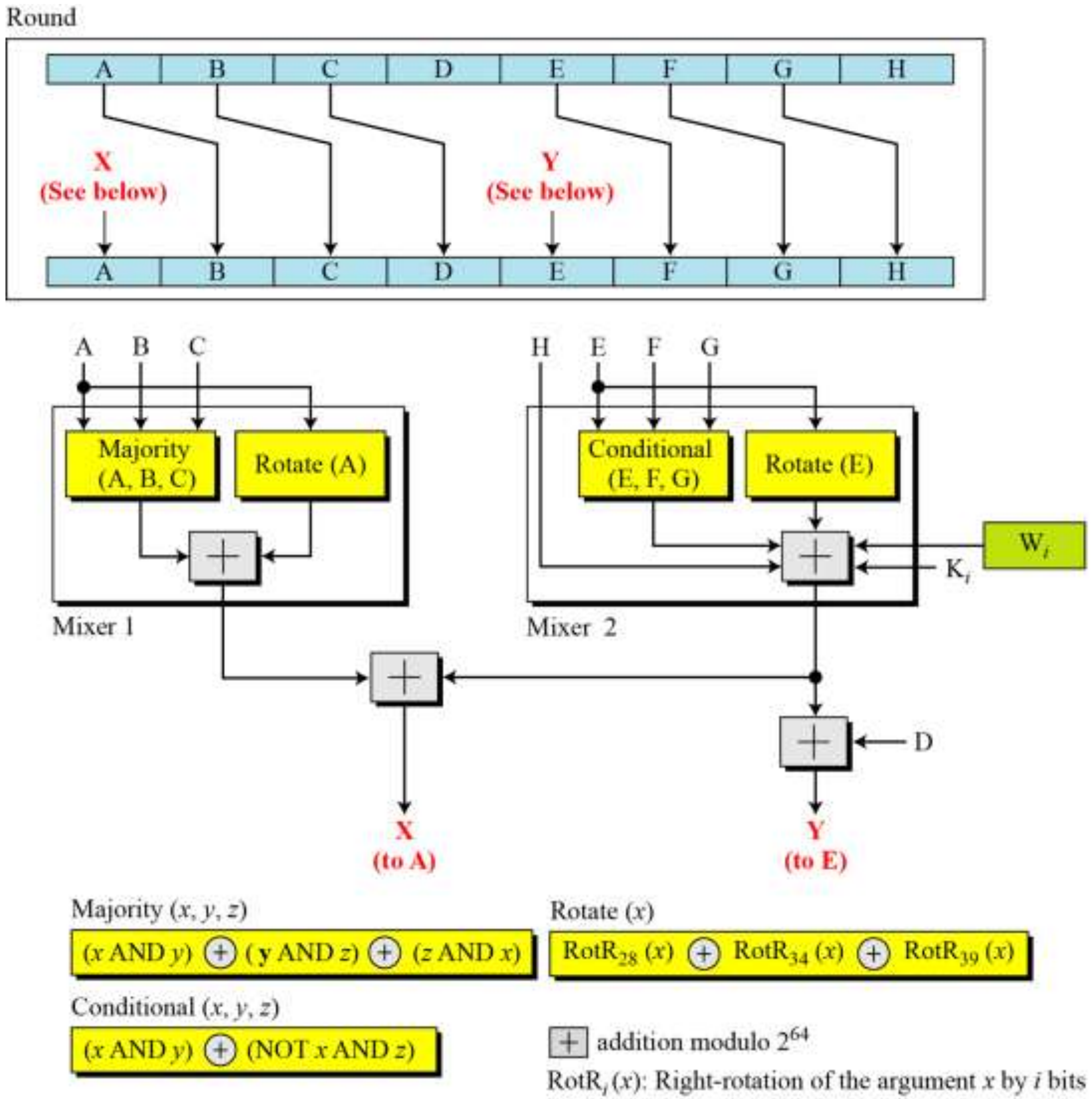
Compression Function



Structure of each round

- In each round, eight new values are for 64-bit buffers are created from the values of the buffers in previous round.
- Six buffers (B,C,D,F,G,H) are exact copies of one of the buffers in previous round.
- Two of the new buffers, A and E, receive their inputs from some complex functions that involve some of the previous buffers, W_i and K_i.
- There are two mixers, three functions, and several operators.

Structure of each round in SHA-512



Majority Function

$$\boxed{(A_j \text{ AND } B_j) \oplus (B_j \text{ AND } C_j) \oplus (C_j \text{ AND } A_j)}$$

Conditional Function

$$\boxed{(E_j \text{ AND } F_j) \oplus (\text{NOT } E_j \text{ AND } G_j)}$$

Rotate Functions

$$\boxed{\begin{array}{l}\text{Rotate (A): } \mathrm{RotR}_{28}(A) \oplus \mathrm{RotR}_{34}(A) \oplus \mathrm{RotR}_{29}(A) \\ \text{Rotate (E): } \mathrm{RotR}_{28}(E) \oplus \mathrm{RotR}_{34}(E) \oplus \mathrm{RotR}_{29}(E)\end{array}}$$

Table 12.3 *Eighty constants used for eighty rounds in SHA-512*

428A2F98D728AE22	7137449123EF65CD	B5C0FBCFEC4D3B2F	E9B5DBA58189DBBC
3956C25BF348B538	59F111F1B605D019	923F82A4AF194F9B	AB1C5ED5DA6D8118
D807AA98A3030242	12835B0145706FBE	243185BE4EE4B28C	550C7DC3D5FFB4E2
72BE5D74F27B896F	80DEB1FE3B1696B1	9BDC06A725C71235	C19BF174CF692694
E49B69C19EF14AD2	EFBE4786384F25E3	0FC19DC68B8CD5B5	240CA1CC77AC9C65
2DE92C6F592B0275	4A7484AA6EA6E483	5CB0A9DCBD41FBD4	76F988DA831153B5
983E5152EE66DFAB	A831C66D2DB43210	B00327C898FB213F	BF597FC7BEEF0EE4
C6E00BF33DA88FC2	D5A79147930AA725	06CA6351E003826F	142929670A0E6E70
27B70A8546D22FFC	2E1B21385C26C926	4D2C6DFC5AC42AED	53380D139D95B3DF
650A73548BAF63DE	766A0ABB3C77B2A8	81C2C92E47EDAEE6	92722C851482353B
A2BFE8A14CF10364	A81A664BBC423001	C24B8B70D0F89791	C76C51A30654BE30
D192E819D6EF5218	D69906245565A910	F40E35855771202A	106AA07032BBD1B8
19A4C116B8D2D0C8	1E376C085141AB53	2748774CDF8EEB99	34B0BCB5E19B48A8
391C0CB3C5C95A63	4ED8AA4AE3418ACB	5B9CCA4F7763E373	682E6FF3D6B2B8A3
748F82EE5DEFB2FC	78A5636F43172F60	84C87814A1F0AB72	8CC702081A6439EC
90BEFFFA23631E28	A4506CEBDE82BDE9	BEF9A3F7B2C67915	C67178F2E372532B
CA273ECEEA26619C	D186B8C721C0C207	EADA7DD6CDE0EB1E	F57D4F7FEE6ED178
06F067AA72176FBA	0A637DC5A2C898A6	113F9804BEF90DAE	1B710B35131C471B
28DB77F523047D84	32CAAB7B40C72493	3C9EBE0A15C9BEBC	431D67C49C100D4C
4CC5D4BECB3E42B6	4597F299CFC657E2	5FCB6FAB3AD6FAEC	6C44198C4A475817

There are 80 constants, K_0 to K_{79}, each of 64 bits. Similar These values are calculated from the first 80 prime numbers (2, 3,…, 409). For example, the 80th prime is 409, with the cubic root $(409)^{1/3}$ = 7.42291412044. Converting this number to binary with only 64 bits in the fraction part, we get

$$\boxed{(111.0110\ 1100\ 0100\ 0100 \ldots 0111)_2 \rightarrow (7.6C44198C4A475817)_{16}}$$

The fraction part: $(6C44198C4A475817)_{16}$

Example

We apply the Majority function on buffers A, B, and C. If the leftmost hexadecimal digits of these buffers are 0x7, 0xA, and 0xE, respectively, what is the leftmost digit of the result?

Solution

The digits in binary are 0111, 1010, and 1110.

a. **The first bits are 0, 1, and 1. The majority is 1.**

b. **The second bits are 1, 0, and 1. The majority is 1.**

c. **The third bits are 1, 1, and 1. The majority is 1.**

d. **The fourth bits are 1, 0, and 0. The majority is 0.**

The result is 1110, or 0xE in hexadecimal.

Example

We apply the Conditional function on E, F, and G buffers. If the leftmost hexadecimal digits of these buffers are 0x9, 0xA, and 0xF respectively, what is the leftmost digit of the result?

Solution

The digits in binary are 1001, 1010, and 1111.

a. **The first bits are 1, 1, and 1. The result is F_1, which is 1.**

b. **The second bits are 0, 0, and 1. The result is G_2, which is 1.**

c. **The third bits are 0, 1, and 1. The result is G_3, which is 1.**

d. **The fourth bits are 1, 0, and 1. The result is F_4, which is 0.**

The result is 1110, or 0xE in hexadecimal.

Analysis

With a message digest of 512 bits, SHA-512 expected to be resistant to all attacks, including collision attacks.